

UNITED STATES PATENT APPLICATION

of

Gilbert M. Wolrich

Mark D. Matson

and

John D. Clouser

for

REDUCTION OF ADD-PIPE LOGIC BY OPERAND OFFSET SHIFT

REDUCTION OF ADD-PIPE LOGIC BY OPERAND OFFSET SHIFT

BACKGROUND OF THE INVENTION

The present invention is directed to floating-point processors and in particular to
5 the addition/subtraction pipelines that they employ.

Computers often use floating-point representations to represent some numbers. And many microprocessors include separate circuitry for performing operations on floating-point numbers. Such circuits take many forms, but the one that Fig. 1 depicts is typical. The floating-point processor 10 depicted there includes a control circuit 11 to which
10 are directed microprocessor operation codes that request floating-point operations. That circuit in turn directs the operation codes that request multiplications, divisions, and additions/subtractions to respective modules sections 12, 13, and 14 that respectively specialize in those operations. Those modules in turn draw their operands from registers 15, in which the modules also store their results.

15 The invention to be described below concerns the addition/subtraction module 14. As is suggested by the drawing's linkage between the division and addition/subtraction modules, the addition/subtraction module may actually be involved in other operations, too, such as assisting in division, converting from fixed-point to floating-point format, or converting between floating-point formats. But for the sake of simplicity the discussion
20 that follows concentrates only on addition and subtraction of single-format floating-point operands and omits any discussion of provisions peculiar to other operations.

Floating-point representations typically have the format < sign, exponent, man-
tissa >. One of the operations that an addition/subtraction module must perform is man-

tissa alignment. Suppose, for example, that operand A is 0.111100×2^{12} and operand B is 0.101000×2^9 . To add those operands' mantissas properly, operand B must be re-expressed as 0.000101×2^{12} ; i.e., its mantissa must be shifted by a number of bit positions equal to the original exponent difference. In that example, the smaller operand's 5 mantissa is shifted by only three bit positions, but an alignment step can involve as much as a fifty-four-position shift in a typical sixty-four-bit machine. Shifts of that size tend to be relatively time-consuming.

For this and other reasons, the addition/subtraction module's process flow may be "pipelined," as Fig. 2 illustrates: the actual addition of one set of operands' (aligned) 10 mantissas may be performed in a third stage 16 concurrently with the alignment in a second stage 17 of the next set to be added—and both may be performed concurrently with an operation in a first stage 18 that determines the amount of shifting required in the set after that. So the addition/subtraction module is often called an addition pipeline, or "add pipe." Fig. 2 depicts the add pipe as comprising three stages, but some add pipes have 15 more or fewer stages.

Now, a floating-point processor's output should be normalized, i.e., so expressed that its mantissa's value is always, say, at least one-half but less than one: the first one bit must occur immediately to the right of the binary point. (This is the VAX floating-point format. In IEEE floating-point formats, the mantissa's value should always be at least 20 one but less than two: the first one bit must occur immediately to the left of the binary point. But the discussion that follows will be based on the assumption that the add pipe employs the VAX format.) So in addition to the just-described, alignment shift, floating-point add pipes also need to perform another, normalization shift. In the previous example, for instance, the raw result of the third stage's mantissa addition is 1.000001, 25 which does not meet the normalization criterion, so the result must be re-expressed as 0.100000 (or 0.100001 after rounding): the mantissa needs to be shifted.

Unfortunately, it is only by performing that addition that it can be determined whether such normalization shifting is necessary in a given instance, and this makes the

normalization shift a critical-path element: the time required to perform it tends to add directly to the total add-pipe latency.

Also contributing to add-pipe latency is the need for rounding. For accuracy, an add pipe's actual adders typically operate with a resolution greater than is permitted in the 5 add pipe's output mantissa. So the add pipe must thereafter round the adders' higher-resolution raw output to the permitted lower output resolution. Although there is more than one way to perform rounding, it typically involves truncating the high-resolution value to the desired low resolution after adding a quantity equal to, say, half the desired quantization interval, i.e., after adding a one bit a single bit position to the right of the 10 position that will be least significant after truncation. But where that position is cannot be determined until the addition has occurred. So rounding circuitry is like circuitry for post-addition normalization: it contributes disproportionately to add-pipe latency, and any improvement that adds to its speed can improve add-pipe performance.

SUMMARY OF THE INVENTION

15 We have found that circuitry for both operations can be made simpler, and thus faster, if a given pair of mantissas' bitwise positions in the addition operation's input for effective additions (i.e., additions of operands whose signs are the same and subtractions of operands whose signs differ) are offset from what they are for effective subtractions (i.e., subtractions of operands whose signs are the same and additions of operands whose 20 signs differ).

Specifically, an effective subtraction's mantissas are applied with a one-bit-rightward offset from the same mantissas' effective-addition positions. As will be explained below, this results in only two choices for round-bit position rather than the conventional three, and it permits the post-addition-normalization circuitry to provide only 25 for one shift direction rather than two.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention description below refers to the accompanying drawings, of which:

Fig. 1, described above, is a block diagram of a representative floating-point processor;

Fig. 2, described above, is a block diagram of such a processor's add pipe;

Fig. 3, described above, is a block diagram of a prior-art add pipe's first stage;

5 Fig. 4 is a block diagram of the second and third stages of an add pipe that employs the present invention's teachings;

Fig. 5 is a block diagram of such an add pipe's first stage;

Fig. 6 is a block diagram of shift-point detector employed in the add pipe's first stage;

10 Fig. 7 is a block diagram of the logic for generating one bit of one of the three alternative outputs of Fig. 6's shift-point detector;

Fig. 8 is a block diagram of the logic for generating one bit of another of the three alternative outputs of Fig. 6's shift-point detector;

15 Fig. 9 is a block diagram of the logic for generating one bit of the remaining one of the three alternative outputs of Fig. 6's shift-point detector;

Fig. 10 is a block diagram of Fig. 5's excess-ones stripper;

Fig. 11 is a block diagram of the circuitry that the add pipe's second stage employs to infer the relative sizes of equal-exponent mantissas' operands; and

20 Fig. 12 is a more-detailed block diagram of the main mantissa adder included in Fig. 4's third-stage circuitry.

DETAILED DESCRIPTION OF AN ILLUSTRATIVE EMBODIMENT

Fig. 4 illustrates the second and third stages of the illustrated embodiment's add pipe. They differ from corresponding stages of similar conventional add pipes in that they are intended to perform multiple-position normalization before any subtraction occurs: the second stage performs the multiple-position normalization shifting on both operands' mantissas before subtraction. This contrasts with arrangements that perform the multiple-position normalization on the single mantissa that results from a first stage's speculative subtraction.

Fig. 4's illustrated second stage accordingly includes not one but two simultaneously operable multiple-position shifters 30 and 32. Like the multiple-position shifter in previous arrangements, shifter 32 is operable to perform both left shifts (for normalization during effective subtraction of similar-magnitude quantities) and right shifts (for alignment during other operations). In contrast, shifter 30 performs only left shifts, since it is used only for normalization.

Shifters 30 and 32 respectively receive minuend and subtrahend mantissas RAW_FRAC_MIN and RAW_FRAC_SUB, which the add pipe's first stage derives from the input-operand mantissas in a manner that will be described presently in connection with Fig. 5. Decoded shift-position vectors (that is, vectors that are all zeros except for at most a single one) L_SHIFT and L_R_SHIFT respectively specify the number of positions by which shifters 30 and 32 shift their inputs. The resultant, shifted mantissas are clocked into latches, not shown but represented by dashed line 33, whose outputs are the third-stage inputs. The add pipe's sole full-width carry-propagate adder 34 adds or subtracts those inputs in accordance with the value of a signal EFF_SUB that the add pipe's first stage has generated to indicate whether the overall operation is an effective-subtraction operation.

To explain the manner in which the first stage generates that signal as well as other inputs to the second stage, we turn to Fig. 5. We assume two input operands OPERAND_A and OPERAND_B clocked into the first stage upon a clock signal. At the same time, the first stage's responses to previous operands are clocked into latches, not shown but represented by dashed line 35, whose outputs are the second stage's inputs. An effective-subtraction-identifying circuit 36 receives those operands' respective sign bits SIGN_A and SIGN_B as well as the operation code that specifies the type of operation the add pipe has been requested to perform. For requested subtractions, circuit 36 asserts EFF_SUB if the sign bits are the same. For requested additions, it asserts EFF_SUB if the sign bits differ.

(Those skilled in the art will recognize that the choice of operation in most add pipes is not merely between addition and subtraction. Also, operation codes typically contain considerably more detail, such as a specification of the floating-point format and the registers from which operands are to be fetched and to which the results are to be stored. But the present invention can be understood more readily without discussing that additional complexity.)

As was explained above, the nature of many of an add pipe's functions depends to a great extent on whether the overall operation is a low-exponent-difference effective subtraction. To determine whether it is, an exponent subtracter 38 compares the operands' exponents EXP_A and EXP_B and applies the result to a circuit 40 that determines whether the magnitude of their difference exceeds one. If not, then an AND gate 42 asserts LOW_EDIFF_SUB during effective subtractions to indicate that the operation is a low-exponent-difference effective subtraction.

If LOW_EDIFF_SUB is not asserted, Fig. 4's multiple-position shifters 30 and 32 should perform alignment shifting. Conventionally, this would mean no shift in the larger operand's mantissa. Fig. 5's multiplexor 43, which receives as inputs the operands' mantissas FRAC_A and FRAC_B, steers the higher-exponent operand's mantissa to shifter 30 as RAW_FRAC_MIN and steers the lower-exponent operand's mantissa to shifter 32 as RAW_FRAC_SUB in response to the exponent difference's sign. This would suggest that left-only shifter 30 should not shift its input, so Fig. 5's multiplexor 44, which generates shifter 30's shift-control vector L_SHIFT, should select its fixed zero-shift-indicating input. The alignment would be achieved by having the left/right shifter 32 shift right by the exponent difference. This would mean that Fig. 5's multiplexors 46 and 48, should forward as shifter 32's shift-control vector L_R_SHIFT the decoded exponent-difference magnitude, which circuit 40 generates as one of its outputs. And, indeed, this is precisely what those multiplexors do when the illustrated circuit performs an effective addition.

During the illustrated embodiment's larger-exponent-difference effective subtractions, though, the asserted output of a further AND gate 50 causes multiplexor 44 to forward a fixed "1"-valued vector as shifter 30's shift-control vector L_SHIFT. This vector's sole one bit is in the position that calls for the larger operand's mantissa to be shifted 5 leftward by a single bit position. And the smaller operand's mantissa is "left shifted" by an additional bit, too, in such operations. Specifically, the assertion of yet another AND gate 52's output, indicating a non-unity-exponent-difference effective subtraction, causes multiplexor 48 to forward the result of right-shifting the decoded exponent difference by one bit position—and thereby decrementing the decoded value by one—instead of forwarding the decoded exponent difference itself. So the smaller operand's mantissa is 10 right shifted by one less position than it would be otherwise.

In short, the bit positions of a given pair of mantissas to be subtracted are one bit to the left of those mantissas' bit positions when they are to be added. We discuss the reason for effective subtractions' extra one-bit leftward shift below in connection with 15 Fig. 12.

Having described the manner in which we develop the multiple-position shifters' shift vectors L_SHIFT and L_R_SHIFT when those shifters are to perform alignment—i.e., when the LOW_EDIFF_SUB signal is not asserted—we now turn to the way in which we develop them when the the multiple-position shifters are to perform normalization. For normalization, the asserted LOW_EDIFF_SUB signal causes Fig. 5's multiplexors 44 and 46 to develop those signals from a (decoded) NORM_SHIFT shift vector, which a normalization-shift-prediction circuit 56 generates, as will be explained below, 20 by comparing corresponding bits of the input-operand mantissas FRAC_A and FRAC_B to predict the position of the most-significant one bit in their difference.

Since the operation being performed is necessarily an effective subtraction—multiple-position normalization occurs only during effective subtractions—the NORM_SHIFT input to the L_SHIFT multiplexor 44's right input port is wired with a 25 one-bit leftward offset to cause our effective-subtraction pre-shift of the larger operand's

mantissa. Fig. 5 depicts this offset as a right-side concatenation of a zero bit. For the smaller-operand-mantissa shift-control signal L_R_SHIFT, the leftward pre-shift is provided by multiplexor 48's selection of a similarly offset input if the exponent-difference magnitude is zero.

5 If the exponent-difference magnitude is one, on the other hand, multiplexor 48 forwards NORM_SHIFT unshifted, thus specifying the required normalization shift without the effective-subtraction pre-shift. The reason for this is that the mantissas need to be aligned when the exponent difference equals one, but Fig. 4's multiple-position shifters would not otherwise align them during low-exponent-difference effective subtractions. So multiplexor 46 in effect adds the one-bit rightward alignment to the one-bit 10 leftward pre-shift, and the two shifts cancel, leaving only the normalization shift.

Actually, Fig. 4's left/right shifter 32 is implemented in a shifter 58 capable of right shifts only, not the left shifts required for normalization. But the add pipe achieves the same result by in effect performing a sixty-four-position leftward shift on the mantissa before applying it to the right shifter 58, which is an extra sixty-four bits wide to accommodate this leftward shifting, and then having shifter 58 shift its resultant contents right by sixty-four minus the desired number of leftward bit shifts.

Specifically, a multiplexor 60 from which shifter 58 receives its mantissa concatenates sixty-four zeroes with the sixty-four bits of the signal that carries the lower-exponent operand's (at most fifty-four-bit) mantissa. If LOW_EDIFF_SUB is deasserted and thereby signals an alignment shift rather than a normalization shift, multiplexor 60 places these zeroes to the mantissa's right, thereby effectively shifting it sixty-four bits to the left. (The shifter's output is its rightmost sixty-six bits, including a round bit and guard bit to the right of the least-significant input bit.) Then, to specify a rightward shift of sixty-four minus the desired number of leftward shifts, the bits of the 20 NORM_SHIFT vector are applied to multiplexor 46 in the order opposite that in which they are applied to multiplexor 44, so multiplexor 46's right input actually represents sixty-four minus the NORM_SHIFT value.

We now turn to a discussion of the manner in which Fig. 5's normalization-prediction circuit 56 determines the required normalization shift. As Fig. 5 shows, circuit 56 includes a shift-point detector 72. A correct prediction of the most-significant one bit's position depends on proper mantissa alignment. Since the proper mantissa alignment is not yet known when circuit 56 begins operation, shift-point detector 72, which Fig. 6 shows in more detail, actually comprises three parallel shift-point detectors 74, 76, and 78 that respectively perform bit-comparison operations for the three alignments possible when the exponent difference is zero or one, and a multiplexor 80 in Fig. 5 selects from among those outputs in response to the output of a two-bit alignment predictor 81.

Each of shift-point detector 72's outputs is a vector that includes, for each mantissa position, a bit that indicates whether corresponding operand bits fit a pattern that could produce the most-significant one bit, or at least the zero bit to the immediate left of the most-significant one bit. Fig. 7 illustrates how each vector bit is generated in the case of the shift-point detector 74 whose output will be selected if operand A's exponent is one greater than operand B's.

In that situation, the i th bit $\text{FRAC_A}[i]$ of operand A's mantissa FRAC_A should be aligned with the $(i + 1)$ st bit $\text{FRAC_B}[i+1]$ of operand B's mantissa, where higher-index bits are more significant. And if $\text{FRAC_A}[i]$ differs from $\text{FRAC_B}[i+1]$, the corresponding bit in the difference between FRAC_A and FRAC_B is a one if the bit position to the right does not generate or propagate a borrow. So AND gate 82 generates a logical-one output when gate 84 indicates that $\text{FRAC_A}[i]$ differs from $\text{FRAC_B}[i+1]$ and gate 86 indicates no apparent borrow in the position to the right, i.e., that a zero value of $\text{FRAC_A}[i-1]$ does not coincide with a one value of $\text{FRAC_B}[i]$.

Actually, a logical one from gate 82 does not necessarily indicate that the mantissa difference will have a one in that position, because actual borrows can be propagated even when there is no apparent borrow in the position to the right. Specifically, a borrow can be propagated to a given position when the operands' bits to the right are the same. In that case, though, the position to the right in the mantissa difference will necessarily be

a one. So a one at a given position in circuit 74's output means that, if operand A's exponent is one greater than operand B's, the mantissa difference has a one either in the corresponding bit or in the bit to the right of it.

If Fig. 5's multiplexor 80 selects circuit 74's output, then an excess-one stripper 88 leaves the most-significant one in that output and removes all ones less significant, so stripper 88's output is a decoded indication of the number of positions by which the operands should be shifted to result in a mantissa difference that is either normalized or requires only a one-position shift before it is.

Inspection of Fig. 8 reveals that its gates 90, 92, and 94 perform the corresponding operation for the situation in which operand B's exponent is one greater than operand A's.

The designs of Figs. 7's and 8's circuits are based on respective assumptions regarding which aligned mantissa will be greater. This assumption can be made reliably because the mantissa of a larger-exponent operand is necessarily larger after alignment. But Fig. 6's shift-point detector 78 is used for the situation in which the exponents are the same, so no such assumption can be made in its design. As Fig. 9 shows, therefore, that circuit's OR gate 96 generates a logical-one output in response to logical-one values of either of two signals DET_AB[i] and DET_BA[i], the former being based on the assumption that operand A's mantissa is greater, and the latter being based on the assumption that operand B's is. Just as Figs. 7 and 8's gates 82 and 90 do, Fig. 9's gates 98 and 100 prevent respective logical-one outputs if respective inverters 102 and 104 report apparent borrows in the mantissas' bit positions to the right. But the other inputs to gates 98 and 100 result from imposing criteria more restrictive than simply that a difference exist between the operand mantissas in the corresponding bit position.

To impose these criteria, the Fig. 9 circuit distinguishes between a "trigger," where the larger mantissa is a one and the smaller is a zero, and an apparent borrow, where the larger mantissa is a zero and the smaller is a one. The two inputs to an OR gate 106 that generates gate 98's other input are generated by gates 108 and 110, which

respectively impose criteria for borrow and trigger positions, as will presently be explained. Similarly, gates 112 and 114 respectively impose criteria for borrow and trigger positions on the inputs that they generate to the gate 116 that generates gate 100's other input.

5 We first consider the criterion that gate 110 imposes on detected triggers. In the mantissa difference that results from the subtraction of one operand from the other, a bit corresponding to a position where the operands form a trigger cannot be the mantissa difference's most-significant logical one if the operand mantissas have a bit difference in the position to the left of that trigger position; a trigger does not propagate a borrow, so that
10 bit difference necessarily results in a mantissa-difference logical one in a position more significant than the one that corresponds to the trigger. Gate 110 therefore forwards its lower input, which is intended to represent detection of a trigger that can result in the mantissa difference's most-significant one bit, only if the operands' mantissas are not different in the bit position to the left.

15 Now, this interpretation of gate 110's output is based on the assumption that operand A's mantissa is larger than operand B's. But if it is operand B that actually has the greater mantissa, then the bit position in question is actually an apparent borrow, not a trigger, and gate 110 asserts its output erroneously. But requiring the operand-mantissa bits not to differ in the bit position to the left makes this error harmless: since the operand-mantissa bits are the same in the position to the left, the borrow makes the mantissa-difference bit in the position to the left a one bit—i.e., the position in question cannot be that of the mantissa difference's most-significant logical one—so the output of gate 100,
20 which is generated under the assumption that operand B's mantissa is greater, will have a logical one at least at one more-significant position, and Fig. 5's stripper circuit 88 will accordingly remove the result of gate 110's erroneous assertion.
25

Gate 108 imposes a related restriction for the borrow-detection case. As was just observed, the mantissa-difference bit in the position to the left of one that corresponds to an apparent borrow is necessarily a logical one if the operand-mantissa bits in that left

position do not differ from each other, so the position corresponding to that apparent borrow cannot be that of the mantissa difference's most-significant logical one. Gate 108 therefore forwards its lower input, which is intended to represent detection of an apparent borrow, only if the operands' mantissas are not different in the bit position to the left.

5 This criterion makes harmless any erroneous assertion by gate 108 of an apparent borrow, for reasons similar to those described above in connection with gate 110.

For the case in which operand B's mantissa is greater, gates 112 and 114 respectively impose the same criteria as gates 108 and 110 do for the case in which operand A's mantissa is.

10 As was mentioned above, Fig. 5's multiplexor 80 generates its possible-shift-point-vector output SHIFTS by choosing among Fig. 6's thus-generated three normalization-shift-point-vector signals, and it applies that output to an excess-ones stripper 88, which, as will now be explained by reference to Fig. 10, removes all but the single most-significant one bit from it.

15 Specifically, each of a plurality of Fig. 10's single-byte strippers 88a-g receives a respective one of SHIFTS's bytes. (Although the drawing depicts eight such strippers to match the floating-point processor's sixty-four-bit-wide data path, the conventional double-precision-format mantissa actually is only fifty-four bits wide.) A respective one of AND gates 117 forwards any one bit in the SHIFTS signal as a corresponding bit of the 20 normalization-shift vector NORM_SHIFT unless another of that AND gate's inputs disables it. A NOR gate 118 receives from each more-significant single-byte stripper an indication of whether that single-byte stripper's input contains a one at any bit position. If so, gate 118's output disables the AND gates 117. Any AND gate 117 is also disabled if any more-significant bit in the same byte is a one bit.

25 As described so far, the operation of Fig. 5's normalization-prediction circuit 56 is essentially the same as that of the corresponding circuit described in the Wolrich et al. patent mentioned above: it sends to the second stage a NORM_SHIFT vector generated by stripping the less-significant logical ones that multiplexor 80 chooses from among the

- shift-point detector 72's three output vectors. But the Wolrich et al. arrangement employs a full carry-propagation adder such as Fig. 3's predictive subtracter 20 to determine the mantissas' relative sizes and thus indicate whether the predictive subtraction's results need to be complemented in a later stage in order to produce a positive output mantissa.
- 5 The arrangement of the Sit et al. patent mentioned above similarly uses a full carry-propagation adder in its first stage to determine which mantissa to subtract from which in order to obtain a positive mantissa.

We avoid such an adder and its attendant delay, but are still able to generate a positive mantissa efficiently, by using a technique that involves forwarding to the second 10 stage not only the NORM_SHIFT vector that Wolrich et al. do but also a DET_AB vector consisting of Fig. 9's DET_AB[i]'s for all mantissa bit positions. In the second stage, Fig. 4's mantissa-comparison circuit 120 uses DET_AB and NORM_SHIFT to determine which mantissa is larger when the exponents are equal. Fig. 11 depicts circuit 120 in more detail. While Fig. 4's shifters 30 and 32 are performing the normalization shift, 15 Fig. 11's AND gates 121 determine whether the bit chosen as the single logical-one bit in the normalization-shift-position vector NORM_SHIFT was the most-significant of the shift bits DET_AB[i] generated on the assumption that operand A's mantissa is greater than operand B's.

If so, OR gate 122 asserts an output FRAC_A_GT_B, which indicates that operand 20 A's mantissa is indeed greater than or equal to operand B's. This means that Fig. 4's minuend shifter 30 has received the correct mantissa, since Fig. 5's multiplexor 43 steers operand A's mantissa to that shifter when the operands' exponents are the same. Otherwise, if the EFF_SUB and EDIFF_EQ_0 signals indicate that the operands' exponents are equal in an effective subtraction, Fig. 4's gate 124 asserts its output INVERT, and the 25 addition/subtraction circuit 34 accordingly subtracts FRAC_MIN from FRAC_SUB, rather than *vice versa*. Consequently, addition/subtraction circuit 34's output is positive, as it is required to be.

The add-pipe output's sign and exponent can be determined in any conventional way. For the sake of example, Fig. 4 includes a multiplexor 130 that determines the output's sign by choosing between the operands' signs on the basis of a signal A_GT_B that indicates which operand is greater in magnitude. In the illustrated embodiment, a multiplexor 132 generates A_GT_B by forwarding the sign output of Fig. 5's exponent subtracter 38 unless the exponent difference is zero, in which case it forwards 5 FRAC_A_GT_B. If a signal NOM_SUB that Fig. 5's circuit 36 generates from the operation code indicates that a nominal subtraction was requested, then Fig. 4's XOR gate 134 first inverts the sign of the nominal subtrahend, operand B. Multiplexor 30's 10 resultant output is the add pipe's sign output SIGN.

Since a de-asserted LOW_EDIFF_SUB signal does not result in the second-stage shifters' performing any normalization, the operation result's exponent should simply be the larger of the two input exponents EXP_A and EXP_B unless normalization occurs further on in the processing chain. So Fig. 5's multiplexor 136 generates a preliminary 15 exponent output signal EXP_AB by selecting between EXP_A and EXP_B in accordance with the sign of the exponent subtractor 38's output, and if the LOW_EDIFF_SUB signal does not indicate that the second-stage shifters performed normalization, EXP_AB is the signal that Fig. 4's multiplexor 138 forwards for further processing to arrive at the exponent output.

A multiplexor 140 forwards that signal further unless the EFF_SUB signal indicates that an effective subtraction is occurring, in which case multiplexor 140 instead forwards the result of an adder 142's decrementing that signal's value to reflect the effective-subtraction pre-shift. And if the final mantissa addition or subtraction results in a carry, a further multiplexor 144 in Fig. 4 selects the result of a further adder 146's incrementing multiplexor 142's output by one. This incrementing reflects the fact that the mantissa adder 34 will perform an additional right shift in that case for normalization purposes, as will be explained below in connection with Fig. 12. 20 25

When LOW_EDIFF_SUB is asserted, on the other hand, the second-stage shifters perform normalization, so the output exponent should be adjusted accordingly. To that end, multiplexor 138 forwards a value that an adder 148 generates by subtracting the size of the normalization shift from the larger-operand exponent EXP_AB. To generate the signal that represents the normalization-shift size, an encoder 150 encodes the normalization-shift vector NORM_SHIFT. Multiplexor 138's output is then processed as before to generate the output exponent EXP.

We now turn to the reason for the extra one-bit leftward shift in the case of effective subtractions. Its purpose is to simplify rounding and final-normalization operations that the mantissa adder 34 performs. Before we describe that simplification in connection with Fig. 12, we consider those rounding and final-normalization operations by simplified examples.

Suppose that an add pipe affording only five-bit output-mantissa resolution is to add $0.11011 \times 2^5 + 0.11101 \times 2^2$, i.e., 27 + 3.625 in fixed-point decimal. Table 1's first two rows represent the alignment of the two mantissas. The completely precise sum, 0.11110111×2^5 , requires eight bits, i.e., more than the add-pipe output's five-bit resolution, so this sum's bits less significant than mantissa A's least-significant, L bit cannot be retained in the five-mantissa-bit output. (In practice, the five mantissa bits would physically be represented outside of the adder by only four bits; since all floating-point quantities are normalized, the first bit, which must accordingly be a one, is only implicit.) For accuracy purposes, though, the adder 34 internally operates on two further bits, the R and G bits. So only one of mantissa B's bits is omitted from the initial addition. Table 1's third row represents the sum of that addition.

Now, if only the resultant bits that correspond in significance to those of the five output bits are considered, the result is 30. But this is not as close to the precise sum, i.e., 30.625, as can be achieved with a five-bit mantissa: $0.11111 \times 2^5 = 31$ is closer. So a common way of rounding is to add half the output quantization interval to the raw sum before truncation, i.e., to add a one in the bit position one less significant than the raw-

sum bit that corresponds to the least-significant output bit, as Table 1's fourth and fifth rows show.

| | C | L | R | G | Decimal | | | |
|-----------|---|---|---|---|---------|---|----|-------|
| A | 0 | 1 | 1 | 0 | 1 | 1 | 27 | |
| B | | | 0 | 1 | 1 | 1 | 0 | 3.625 |
| Raw Sum | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 30.5 |
| Round Bit | | | | | 1 | 0 | | |
| Rounded | | | 1 | 1 | 1 | 1 | 1 | 31 |

TABLE 1

We note here that this is not the only rounding mode that high-performance add pipes typically implement. Another mode, known as “round to nearest even,” is the same as the approach just described, with one exception. If mantissa B in that example had been exactly 3.5 before truncation, so that the exact sum would have been 30.5 and thus equidistant between 30 and 31, then the round-to-nearest-even approach would round to 30 (the “nearest even”) rather than 31, as the illustrated approach would. The round-to-nearest-even approach sets the L bit to zero if “sticky bit” circuitry, which records the significance of the least-significant one bit in the smaller-exponent operand's pre-truncation mantissa, indicates that no one bits were discarded during truncation. Another mode, whose purpose is to round to the next higher quantization level, usually adds a full quantization interval before truncation rather than half one, but only if the overall addition's sign will be positive. And there are still further approaches, too.

To avoid unnecessarily complicating the discussion, we assume only a single rounding mode here and in the Fig. 12 discussion. But other modes also share a complication that will now be illustrated by reference to Table 2's illustration of that single mode. Specifically, the bit position at which the round bit needs to be added depends on the raw sum. In the Table 2 example, the addition produces a carry, and the result of the rounding will not be satisfactory if the round bit is again added in the R-bit position. Table 2's fourth through sixth rows show the result of (inappropriately) doing so. Because there is a carry, a rightward normalization shift is necessary, and the resultant output is 32, whereas the infinite-precision result would be 33.375. Although a five-bit

mantissa cannot represent the quantity 33, it can represent 34, which is still closer to the infinite-precision result than Table 2's output.

| | C | L | R | G | Decimal | |
|------------|----------|----------|----------|----------|----------|----------------|
| A | 0 | 1 | 1 | 1 | 0 | 30 |
| B | | 0 | 1 | 1 | 0 1 | 3.375 |
| Raw Sum | 1 | 0 | 0 | 0 | 1 0 1 | 33.25 |
| Round Bit | | | | | 1 | |
| Rounded | 1 | 0 | 0 | 0 | 0 1 1 | |
| Normalized | 1 | 0 | 0 | 0 | 0 | 32 (÷2) |

TABLE 2

To deal with this problem, rounding adders conventionally inspect the initial addition's carry bit and add the round bit in the L-bit position if that carry bit is set. Table 3 illustrates thus adding the round bit in the proper, L-bit position.

| | C | L | R | G | Decimal | |
|------------|----------|----------|----------|----------|----------|----------------|
| A | 0 | 1 | 1 | 1 | 0 | 30 |
| B | | 0 | 1 | 1 | 0 1 | 3.375 |
| Raw Sum | 1 | 0 | 0 | 0 | 1 0 1 | 33.25 |
| Round Bit | | | | | 1 | |
| Rounded | 1 | 0 | 0 | 0 | 1 0 1 | |
| Normalized | 1 | 0 | 0 | 0 | 1 | 34 (÷2) |

TABLE 3

Tables 2 and 3 also illustrate another sum-dependent aspect of floating-point processing: if a carry occurs, the adder must perform a post-addition normalization shift to the right.

Conventionally, moreover, the choices are not just between not shifting and shifting to the right. If the operation is a conventional effective subtraction, the direction of the post-addition normalization must be to the left, as Table 4 illustrates.

| | C | L | R | G | Decimal | | |
|------------|---|---|---|---|---------|------------------|--------|
| A | 0 | 1 | 0 | 0 | 1 | 17 | |
| B | | | 1 | 0 | 0 | 1 | 2.375 |
| 2's Compl. | 1 | 1 | 1 | 1 | 0 | 1 | -2.375 |
| Raw Sum | 0 | 0 | 1 | 1 | 1 | 0 | 14.5 |
| Round Bit | | | | | 1 | | |
| Rounded | 0 | 0 | 1 | 1 | 1 | 0 | 14.5 |
| Normalized | | | | | | 14.5 (x2) | |

TABLE 4

(Table 4's third row represents the two's complement of mantissa B's complete-precision value, not the two's complement of the truncated value that the mantissa adder receives. Obviously, the mantissa adder cannot add a one bit at the complete-precision value's least-significant bit position, as is necessary to two's-complement the illustrated mantissa with complete precision. But it can obtain the truncated version of that result by using well-known "sticky bit" circuitry, not shown in the drawings, that notes whether any one bits were discarded during truncation and, if so, suppresses the mantissa adder's addition of a one to the G-bit position during the two's-complementing process. This is equivalent to adding the one bit at the least-significant position before truncation.)

Table 4 also shows that the round bit conventionally must be added in the G position to achieve 14.5, which is the five-bit number closest to the exact answer of 14.625. As Table 5 shows, the result would be 15.0 rather than the more-accurate 14.5 if the round bit were instead added in the R position.

| | C | L | R | G | Decimal | | |
|------------|---|---|---|---|---------|------------------|-------|
| A | 0 | 1 | 0 | 0 | 1 | 17 | |
| B | | | 1 | 0 | 0 | 1 | 2.375 |
| 2's Compl. | 1 | 1 | 1 | 1 | 0 | 1 | -2.25 |
| Raw Sum | 0 | 0 | 1 | 1 | 1 | 0 | 14.75 |
| Round Bit | | | | | 1 | | |
| Rounded | 0 | 0 | 1 | 1 | 1 | 0 | 15.25 |
| Normalized | | | | | | 15.0 (x2) | |

TABLE 5

But because the bit positions that Fig. 4's shifters 30 and 32 produce for effective subtractions is one bit to the left of the positions that they produce for effective additions,

the mantissa adder 34 never needs to add the round bit in the G position or to shift its results to the left for normalization.

To appreciate the resultant simplification, consider Fig. 12's simplified representation of a high-performance mantissa adder. A lowest-nibble adder 160 and dual byte adders 162a, b, ..., g (collectively, "162") generate the actual sum bits by operating on the results of some preprocessing of the minuend and subtrahend mantissas FRAC_MIN and FRAC_SUB.

These adders speed the addition process by breaking the addition into nibble- or byte-sized segments. Recall that the sum and carry bits from a given bit position of a full adder depend not only on the corresponding two input bits but also on the carry out of that adder's bit position to the right, which in turn depends in part on the carry out of one further bit to the right, and so on. To avoid the delay that this propagation of carry results from low bits to high bits could cause in the addition of, say, fifty-four-bit mantissas, the dual byte adders 162 do not wait for the lower-significance adders' carry results. Instead, they perform two additions in parallel, one producing a sum S_{CI} based on the assumption of a carry in from the next-most significant byte adder and the other producing a sum S_{NCI} based on the assumption of no such carry.

Multiplexors 164 a, b, ..., g (collectively, "164") then choose the correct alternative result when it is known where carries in have occurred. They are depicted as four-to-one multiplexors because they not only make that selection but also perform post-addition normalization by selecting between unshifted and right-shifted versions of the correct alternative. The manner in which this normalization is performed will be explained after a discussion of the way in which the add pipe chooses the correct alternative sum S_{CI} or S_{NCI} .

We begin that discussion with the lowest-nibble adder 160. This adder actually adds the round bit, generated by a multiplexor 166, to the raw sum's lowest nibble, generated by a single nibble adder 168. If adder 168's carry output 170 indicates that there is a carry out of the raw sum's lowest nibble, an OR gate 172a asserts one of the select in-

puts of multiplexor 164a, which responds by choosing the sum S_{CI} that adder 162a generated by assuming that there would be a carry out of the lowest nibble.

Note that adder 168's output is the low nibble of the raw sum, i.e., of the sum before rounding. This means that there could be a carry out of the low nibble without output 170's necessarily being asserted, yet S_{CI} would still need to be chosen. When this happens, OR gate 172a's other input is asserted, as will be explained below, and multiplexor 164a still makes the correct selection.

Adder 168's carry output 170 is also the basis for adder 162a's generation of its predicted-carry output 174, which adder 162b uses in generating its predicted-carry output 176, and so on until the predicted-carry input 178 of the highest-significance adder 162g indicates whether its byte receives a carry in computing the raw sum. If predicted-carry output 174 or 178 indicates a carry, respective OR gate 172b or 172g asserts the associated select input of respective multiplexor 164b or 164g to choose the respective S_{CI} value. If a given OR gate-172 output is unasserted, then the respective multiplexor forwards the respective S_{NCI} value instead.

The reason why carry signals 174, 176, and 178 are referred to as "predicted" is that each is actually generated in parallel with part of the carry calculation that enters into its associated adder's S_{CI} and S_{NCI} generation. A single-bit segment of a full adder can be thought of as consisting of two stages. The first stage's output indicates whether the corresponding two addend bits (1) are both ones and will therefore produce a carry, (2) are a one and a zero and will therefore propagate any carry received from the bit position to the right, or (3) are both zeroes and will therefore "kill" such a carry. All bit positions' first stages can operate completely in parallel because the output of each depends only on the two corresponding operand bits. The illustrated embodiment's byte adders generate the carry-predict outputs in response to their first stages' outputs by employing circuitry that operates in parallel with the adders' second-stage circuits.

A given bit position's second stage is what generates that bit position's sum and carry results. In doing so it uses not only the corresponding first stage's generate/-propagate/kill output but also the carry from the second stage of the bit position to the

right: carries must ripple through successive bit positions. Instead of waiting for the second stages' rippling to determine whether there will be a carry out of a given byte, the carry-predict circuitry responds to the byte adder's constituent individual-bit generate/-propagate/kill outputs by producing group generate/propagate/kill outputs that indicate whether the byte adder will generate a carry, propagate a carry from the previous adder if one is received, or kill such a received carry. Then, when the predicted-carry input from the previous adder becomes valid, the carry-predict circuitry determines its predicted carry from that input and the group generate/propagate/kill signals that it has already produced. It turns out that having OR gates 172 use the resultant outputs in determining which of the adders' sum outputs are to be selected is faster than basing that determination on the second stages' carry outputs.

But gates 172 also need an input that predicts whether adding the round bit will cause carries not produced by the raw sum's computation; the carry predictions just described are based only on the raw sum, not the rounded sum. For a given byte to receive such a carry, the rounded sum's lowest nibble must produce a carry out, and the adders associated with all bytes less significant than the byte in question must generate carries out if they receive carries in.

Whether the rounded sum's lowest nibble will produce a carry out depends on where the round bit is to be added. This in turn depends on whether there will be carry from the highest-significance byte adder 162g. If that adder's output C_{NCI} is asserted, indicating that it will generate a carry even if there is no carry in, then an OR gate 180's output MSB_BR is asserted. This indicates that before rounding the raw sum's most-significant bit, i.e., the bit in the carry position, is a one and that a rightward normalization shift will be needed. An AND gate 182's output also causes MSB_BR to indicate the need for a normalization shift if predicted-carry input 178 indicates that adder 162g's byte receives a carry in before rounding and adder 162g's C_{CI} output indicates that such a carry in will cause a carry out.

As was explained above, a number of considerations can enter into where and if the round bit is to be added. In Fig. 12's simplified representation, though, it is assumed

that the only round mode employed is the one in which a round bit is always added and in which the location of its addition depends only on the normalization that the mantissa adder performs. In that mode, MSB_BR's assertion causes multiplexor 166 to choose its 010 input, which in turn causes adder 160 to add a one to the L bit of the raw-sum low-nibble value. (Multiplexor 166's three-bit output is the left three bits of adder 160's right input.) MSB_BR's assertion also gates to one input port of an OR gate 184 a C_H signal, which is asserted if the pre-rounding low nibble's L bit and its most-significant, K bit are both ones. If they are, then adding the round bit at the L position will cause a carry, and OR gate 184 asserts its RND_CARRY output, indicating that the low nibble generates a carry out after rounding. If MSB_BR is not asserted, then multiplexor 166 chooses its 001 input, thereby causing the round bit to be added at the R bit position and producing an otherwise-absent carry only if the K, L, and R bits are all ones. If they are, as indicated by an asserted C_L signal, then OR gate 184 asserts its RND_CARRY output even if MSB_BR is not asserted.

For adder 162a's carry-in output sum S_{CI} to be multiplexor 164a's proper selection even if the raw-sum computation produced no carry into adder 162a, it is necessary only that adding the round bit produce an otherwise absent low-nibble carry. This condition is what RND_CARRY's assertion represents, so it causes OR gate 172a to assert its output and multiplexor 164a to select adder 162a's carry-in output sum S_{CI}.

For a higher-significance adder's S_{CI} output to be the proper selection in the absence of a raw-sum carry, the second condition is that all lower bytes produce carries out if they receive carries in. To determine whether both conditions prevail for adder 162b, an AND gate 186b receives the RND_CARRY signal and the previous byte adder 162a's C_{CI} output. Simultaneous assertion of both signals indicates that the round bit's addition will cause a carry into adder 162b's byte, and the resultant assertion of gate 186b's output causes multiplexor 164b to choose the sum output S_{CI} that adder 162b generated by assuming that carry in. Similarly, AND gate 186g causes multiplexor 164g to choose adder 162g's S_{CI} output if the RND_CARRY signal and all previous byte adders' C_{CI} outputs are asserted.

Together with a further multiplexor 188, multiplexors 164 also shift the selected sum segments to the right if normalization is required. Suppose, for example, that OR gate 172a asserts its output, thereby indicating to multiplexor 164a that the associated adder 162a's carry-in sum S_{CI} should be used. If that multiplexor's other selection input, 5 MSB_BR, is not asserted, then multiplexor 164a selects its leftmost input 190. But if MSB_BR is asserted, indicating the need for a normalization shift, then multiplexor 164a instead selects input 192. That input consists of the seven high bits of adder 162a's carry-in sum S_{CI} and, concatenated to their left, multiplexor 194b's output, which is the least-significant bit of adder 162b's selected output.

10 In theory, such a shift should occur not only when MSB_BR is asserted, indicating a set carry bit before rounding, but also when the carry bit is set only after rounding, in which case MSB_BR would be unasserted. But the shift is unnecessary in practice, since the normalization in that case is always from 1.000... to 0.100...; it merely shifts the sole one-valued bit into the "hidden bit" position, which outside the floating-point 15 processor is only implicit. So if the carry occurs only after rounding, the output is all zeroes regardless of whether the output multiplexors 164 and 188 perform a normalization shift.

Still, the exponent must be incremented in that case. So when gate 172g's output indicates that the most-significant-byte adder 162 is to receive a carry in and that adder's 20 S_{CI} output indicates that a carry out will result from a carry in, an AND gate 196 sends Fig. 4's multiplexor 144 an asserted CARRY signal. Thus causes it to select the incremented exponent value.

It was mentioned above that the inputs to adders 162 and 168 result from initial processing of the mantissa adder's input FRAC_MIN and FRAC_SUB. When the operation 25 is an effective subtraction, as indicated by an asserted EFF_SUB signal, one part of that initial processing is two's complementing of the subtrahend: subtraction is performed by adding the subtrahend's two's complement to the minuend. To take the two's complement of a number, that number is bitwise complemented and a one bit is added at the result's least-significant bit position. Fig. 12 represents the one-bit addition by

EFF_SUB's concatenation with multiplexor 166's round-bit output to produce the value that adder 160 adds to the raw sum's low nibble. (For the sake of simplicity, we omit the sticky-bit circuitry, mentioned above in connection with Table 4, which causes concatenation of a zero instead of a one if any one bits were discarded in the alignment process.)

In most effective subtractions, FRAC_SUB is to be subtracted from FRAC_MIN, so it is FRAC_SUB that is complemented: XOR gates 198 ordinarily complement FRAC_SUB when EFF_SUB is asserted. But when the INVERT signal indicates the FRAC_MIN is actually the smaller mantissa, FRAC_MIN should be subtracted from FRAC_SUB, so gates 198 pass FRAC_SUB uninverted, and gates 200 invert FRAC_MIN.

Half adders 202 perform the other part of the initial processing. The half adders' purpose is to eliminate a problem that results from the fact that a rounding adder actually adds three numbers rather than just two: it adds not only the two mantissas but also a value represented by the round bit. Now, when two binary numbers are added, there are only two carry possibilities at a given bit position: either a one is carried out of that position or it is not. The dual adders 162 are based on this assumption: they produce only two alternative sums. In the absence of the preprocessing, though, the addition of three numbers would add a third possibility: a carry out of 10_2 , as Table 6 illustrates.

| | Low Nibble | | | | Decimal |
|----------------|------------|---|---|---|---------|
| | K | L | R | G | |
| FRAC_MIN | | | 1 | 1 | 3 |
| FRAC_SUB | | | 1 | 1 | 3.75 |
| Round Bit | | | 1 | 0 | 1.25 |
| Low-Nibble Sum | | | 0 | 0 | 0 |
| Carry out | 1 | 0 | | | |
| Total | 1 | 0 | 0 | 0 | 8 |

TABLE 6

This would necessitate the first byte adder 162a's being a triple byte adder rather than a dual byte adder, and it would complicate the output multiplexing and selection. To avoid such complications, the half adders convert the two input mantissas into a different pair of mantissas that have the same sum. Specifically, FRAC_MIN and FRAC_SUB are

converted to FRAC_X and FRAC_Y, where each bit of FRAC_X is the corresponding bit of the half adders' sum output and each bit of FRAC_Y is the (half-adder) carry from the previous bit position:

$$\text{FRAC}_X[i] = \text{FRAC_SUB}[i] \text{ XOR } \text{FRAC_MIN}[i]$$

5 $\text{FRAC}_Y[i] = \text{FRAC_SUB}[i-1] \text{ AND } \text{FRAC_MIN}[i-1]$

It turns out that such a conversion eliminates the extra-carry problem, as Table 7 illustrates, by shifting the value of one of the carries into the byte to the left before the raw addition occurs.

| | | Low Nibble | Decimal |
|----------------|-----|------------|---------|
| | | L R G | |
| FRAC_MIN | | 1 1 | 3 |
| FRAC_SUB | | 1 1 1 1 | 3.75 |
| FRAC_X | | 0 0 1 1 | 0.75 |
| FRAC_Y | 1 | 1 0 0 0 | 6 |
| Round Bit | | 1 0 1 | 1.25 |
| Low-Nibble Sum | | 0 0 0 0 | |
| Carry Out | 1 | | |
| From Next Byte | 1 | | |
| Total | 1 0 | 0 0 0 0 | 8 |

TABLE 7

10 Perusal of Fig. 12 in light of the foregoing discussion reveals the simplification that the effective subtraction pre-shift provides. First, only two alternatives are necessary for round-bit addition, so the circuitry represented by multiplexor 166 can be significantly simpler than it needs to be in conventional arrangements. The logic needed to develop its selection signals is simpler, too.

15 More important, the circuitry that multiplexors 164 and 188 represent can be less complicated, and thus faster, than corresponding circuitry in conventional arrangements. Specifically, they need to select between only two shift positions, not three. In some implementations, this makes it practical to include the shift selection carry/no-carry selections in the same multiplexor, as Fig. 12 illustrates, rather than in a subsequent stage that adds further delay.

Moreover, the simplification that the present invention affords is even greater in most high-performance add pipes than is apparent in Fig. 12. As was mentioned above, that drawing omits much of the more-complicated circuitry needed to provide more than one rounding mode. Much of that omitted circuitry, too, can simplified if the present invention's pre-shifting is employed.

It should also be noted that, although we prefer to perform a leftward pre-shift for effective subtractions, most of the invention's advantages can be obtained by instead performing a rightward pre-shift for effective additions. In that case the round-position choice would be between the R and G positions instead of the L and R positions, and the normalization-shift direction would be left rather than right. Also, although we prefer to include the main adder 34's normalization shift in the same operation as the individual adders' sum selection, it could instead be performed in a subsequent operation (or, less advantageously, a previous one). And there is no reason why the pre-shifting needs to be performed by the same circuitry that does the multiple-position shifting. It could instead be performed together with the mantissa selection that Fig. 5's multiplexor 43 represents, for example.

In short, the present invention's teachings can be implemented in a wide range of embodiments, which can differ significantly from the one illustrated above. It thus constitutes a significant advance in the art.

20 What is claimed is: